①

Performance Evaluation of the MOS Distributed System

Richard Wheeler

Thesis Submitted Towards the
Degree of Master of Science

Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel 91904

July 1988

האוניברסיטה העברית בירושלים
המחלקה למדעי המחשב

THE HEBREW UNIVERSITY OF JERUSALEM
Department of Computer Science
Jerusalem, Israel

1

Performance Evaluation of the MOS Distributed System

Richard Wheeler

Thesis Submitted Towards the
Degree of Master of Science

Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel 91904

July 1988

ABSTRACT

The MOS system is an integrated multicomputer system which was designed to preserve the standard UNIX interface while providing complete network transparency. This thesis measures and analyses the performance of the internal mechanisms of the system, including all of the system calls, interprocess communication mechanisms and process migration. Several distributed application programs, the most successful of which show an almost linear improvement in performance as the number of processors increases, are also analyzed.

Acknowledgement

Many thanks to Professor Amnon Barak for his patient guidance and critical suggestions. This work was supported in part by the U.S Air Force Office of Scientific Research under grant AFOSR-85-0284, in part by the National Council for Research and Development, grant no. 2525, and in part by the Foundation for Research in Electronics Computers and Communication, Israel Academy of Science and Humanities. Thanks to Omri Mann for his help in implementating the Travelling Salesman Problem.

Chapter 1
Introduction

Distributed systems provide transparent access to remote objects, *e.g.* files or programs, and allow user applications to simultaneously utilize the processing power of all of a system's processors. For a distributed system, the performance of remote operations, *e.g.* remote procedure calls or a memory mapping mechanism between local and remote machines, determines the feasibility of the distributed system itself and the feasibility of other mechanisms. Measuring the performance of a system's mechanisms has two benefits: it highlights the weak points of the system for the system designers and it provides the data needed by users to predict the performance of applications which they want to run on the system. This thesis deals with measuring and analyzing the kernel mechanisms of the MOS multicomputer operating system and several application programs which run on MOS.

MOS [1-6] is a Multicomputer Operating System which integrates a cluster of loosely connected computers into a single machine UNIX environment. The system uses decentralized mechanisms to provide a network transparent file system, dynamic process migration and load balancing. The kernel of MOS is divided into three parts: the *upper kernel*, the *linker* and the *lower kernel*. The upper kernel provides users with the standard UNIX interface. The lower kernel contains all of the routines which handle local resources. Eventually, the upper kernel calls the relevant lower kernel procedure via the *linker* which determines whether the call can be executed locally or needs to be sent to a remote machine. Remote system calls which need to return data other than the return value use the *funnel* mechanism which maps memory from a local machine to a remote one.

System calls define the basic interface between the user's programs and the operating system. In MOS, system calls can be divided into three groups: system calls with no remote implementation, system calls with remote versions which use the *funnel* mechanism and system calls with remote versions which do not use the *funnel* mechanism. In Chapter 3, benchmarks are used to determine the slowdown associated with each remote kernel system call: computed as the elapsed time required for the remote execution of the call divided by the elapsed time required for the local execution. It should be noted that the design of the MOS kernel allows for certain system calls to be executed locally regardless of the current position of the calling process. We present results which show the weighted average (by frequency of the system calls) to be 240%. The same technique was used by Leffler *et al* and McKusick *et al* [7, 8] to measure the performance of system calls of Berkeley UNIX systems and by Cabrera *et al* [9, 10] to analyze the network throughput of IO related system calls on a Berkeley UNIX system. While comparing results gained on different machines is of dubious value, in Douglis and Ousterhout [11] the measured slowdown of process related remote system calls in the Sprite system was approximately twice that of the corresponding MOS system call.

In Chapter 4, we measure the slowdown factor associated with the interprocess communication (*IPC*) related system calls. The cost of the IPC is a critical factor in determining the performance of distributed applications. The IPC mechanisms used in MOS show a similar slowdown factor to the general system calls. The measurements proved the importance of using a distributed IPC mechanism: centralized mechanisms, like the current initial of AT&T System V messages, which store messages on one machine, create a serious I/O bottleneck which hinders the scalibility of distributed applications.

A unique feature of MOS is its dynamic process migration by the automatic load balancing mechanism. In Chapter 5, we describe the performance of the implementation of the *funnel* mechanism since it is the basis of the process migration mechanism and then the speed of process migration itself. The best result obtained showed the *funnel* throughput to be 164 Kbytes/second. The results presented in [10] show the maximum network throughput over ethernet to be 188Kbytes/second with an unloaded VAX11/780, a processor almost twice the speed of the MOS MC68010 based node, and an unloaded Ethernet.

In Chapter 6, the speedup factor of distributed applications is measured when run on the MOS system. The speedup factor is computed to be the minimum run time the application requires on a one processor system divided by the run time on an n processor system. An application which does a large amount of IO (each operation requiring a remote system call) is more difficult to distribute since each of the system calls suffers a slowdown. On the other hand, an application which does heavy computation, does not perform many system calls and does not show a degradation in performance on a remote node. For example, an application which spends 9 seconds running in user mode and 1 second system time can expect to gain a speedup of at best (9 / number of processors) + 2.4, where the measured slowdown of system calls is 240%. The first application, a parallel version of the UNIX *make*, does a large amount of disk IO as well as computation. By distributing the temporary files created by each of its subtasks between the working nodes at random, a speedup of 60% of the optimal was measured. The second application is an distributed implementation of the Traveling Salesman Problem [12], a CPU bound problem that does relatively little message passing. For all but the smallest instances of the problem, a nearly linear speedup was obtained (the speedup with 7 machines being 88% of the optimal).

This thesis is organized as follows. Chapter 2 describes general properties of distributed systems and gives a overview of the kernel structure of the MOS system. Each of chapters 3 through 5 describes the performance of one mechanism of the MOS system. Chapter 6 describes the performance of two application programs and Chapter 7 draws conclusions.

Chapter 2
The MOS Multicomputer System

2.1. Introduction

An integrated multicomputer system is a collection of computers which run a single, homogeneous operating system and can communicate through a common network. The basic goal of such systems is providing each computer access to non-local resources, such as files or remote processing power. Some such systems, like Locus [13], are composed of heterogeneous machines. The systems discussed in this paper are *homogeneous* systems in that all of the machines (*nodes*) are required to have the same instruction sets. n this section, only systems which support the UNIX interface are used for comparison.

2.2. Typical Features of Distributed UNIX Systems

The above definition of multicomputer systems is quite general and includes a multitude of different types of systems. In practice, distributed systems can be identified as systems which provide some subset of the features described in this section. Later chapters analyze the MOS distributed operating system's implementation of these features.

Network file systems provide a system's users access to remote files and have become increasingly popular in recent years. Most UNIX based distributed file systems provide user transparent access to remote files. That is, the user need not be aware of the location of a file as long as he or she knows the UNIX path to the file which is a system wide identifier. In UNIX, the kernel procedure *namei* maps a pathname to a unique file header, called an *inode*, which contains a description of the file and its location. The *namei* procedure must be modified in a distributed system to handle the remote file names. Some systems employ a universal *super-root* to tie the various file systems into a single tree while others provide arbitrarily placed *remote namespace escapes*, which point to the inode of some directory on a remote machine. This turns the standard UNIX tree into a general network wide graph. For a more comprehensive discussion of network file systems see [6].

Distributed systems have several mechanisms for allowing processes to communicate across the network. Remote procedure calls (*RPC*'s) are an extension of general procedure calls which allow a process to execute certain procedures on a remote node. The advantage of a RPC is its ease of use: no change in programming style is required, except for the change in syntax required to specify where to run the procedure.

Message passing, the second mechanism, is a more radical break from standard programming in that messages allow multiple, concurrent threads of execution. Programmers are provided with a set of procedures to send and receive messages. Message passing is a central mechanism in server/client based systems and has a significant influence on the performance of such systems [14]. Both of the above mechanisms provide some access to the processing capabilities of remote nodes.

The *throughput* of a distributed system, *i.e.* the number of job completions in a given period of time, can be improved by *load balancing*. Load balancing algorithms can be divided into two major classes: static algorithms and dynamic algorithms. Static load balancing algorithms assign a new process to a processor upon its creation. The process continues executing on this node regardless of how the load distribution changes during is life time. Dynamic load balancing algorithms allow a process to *migrate* to a less loaded node if the distribution of the work load changes during the execution of the process. In load balancing systems, the performance of the various distributed mechanisms is taken into consideration when deciding whether to migrate a process from its original node. A detailed survey of various load balancing schemes is given in [15].

2.3. The MOS System

The MOS system [1-6] is a loosely coupled, integrated multicomputer system which was designed to preserve the standard UNIX interface while providing complete network transparency. Each node is completely autonomous and supports the classic UNIX interface. The amount of memory, processor speed and the number and type of peripherals may vary from one node to the next. The MOS configuration that was used in the current work is a collection of Cadmus QU68000's which are based on the Motorola MC68010 processor and are connected by Pronet, a 10 Mbits/second token passing ring. In addition, each machine has one megabtye of main memory and an 80 megabyte disk.

2.3.1. MOS System Architecture

The kernel of the MOS operating system is divided into three parts: the *lower kernel*, the *linker* and the *upper kernel*. The lower kernel contains device drivers for the local node and all of the routines which access file and process structures. It is tightly coupled to the local node and can access only local objects or objects that have migrated to the local node. The upper kernel provides the standard UNIX system call interface. When a process executes a system call, the upper kernel performs the preliminary processing of the parameters, *e.g.* calling *namei* to parse a path name into a ***universal inode*** or checking that the user has permission to access a certain object. Eventually, the upper kernel calls the relevant remote kernel procedure (or *scall*) to complete the service. For example, the call **Sproc_name(machine_id, param_list)** invokes the remote kernel procedure "proc_name" with the parameters specified in "param_list" on machine "machine_id".

At this point, the Scall is passed to the linker which decomposes the *scall* into the procedure name and the parameter list. The linker examines the parameters of the call to determine if the call needs to be executed remotely. If it is remote, the linker encapsulates the call into a message and sends the message over the network. If the call can be executed locally, the linker invokes the local lower kernel procedure directly. On the target machine, an *ambassador process*, a lightweight kernel process, executes the appropriate lower kernel procedure for the calling process. The result of the system call is then encapsulated into another message and returned to the calling node. Remote system calls which need to transmit large amounts of data in addition to the return value of the RPC use the MOS *funnel* mechanism to copy the data from the memory space of one machine to another. The linker handles the implementation of funnels by breaking up large blocks of data into message size pieces at one end, sending the messages over the network and then reunifying the data in the proper order on the target machine. The data is then copied by the linker of the receiving node to the specified memory address.

For example, for the *read* system call, the upper kernel sets up an input funnel on the local machine before calling the *Sread* remote system call. If the system call accesses a remote file, the linker routes the call to the target machine and an ambassador process there calls the appropriate kernel procedure for reading a file. As each logical file block is read, the data is placed into the remote end of the funnel and passed back to the initiating machine's linker. After the call completes, the returned status of the system call is encapsulated by the remote linker and passed back to the calling machine.

2.3.2. The MOS File System

The MOS file system is based on the standard UNIX file system tree. The original MOS file system used a super root, "/...", as a network wide root. When addressing a file on a remote machine, the user prefixes the super root to the machine name and then adds the usual UNIX path. For example, "/.../m2/etc/passwd" is the absolute path name for the password file on machine number two. In the version of MOS measured in this work, a new scheme uses a special file type which indicates the remote machine number in its inode. When such an inode is accessed by the kernel, the inode of the special file is automatically replaced with the inode of the root directory of the remote node indicated. For example, if the special file "/usr/systems/bert" is created as a remote escape to the file system on machine two, the path "/usr/systems/bert/etc/passwd" refers to the password file on machine two. Note that this change eliminates the need for special, non-standard UNIX pathname syntax. Also, it allows the MOS file system to have arbitrarily placed links between nodes.

In MOS, the inode of an open file is held by the site at which the file resides. All remote *opens* are returned a universal pointer to the inode, a *universal inode*, which is used for future file accesses. A garbage collection algorithm is used to clean up allocated inode structures in the case of a failure of the host node or of the calling node.

In order to allow for dynamic reconfiguration of the system, *e.g.* the removing or adding of new machines at any given point in time, each remote access generates a new remote kernel call which returns a special error code if the node is currently unreachable. The system does not require any special action to incorporate a new node: upon the first remote call to the newly joined node, the system simply sends the message to it in the usual way and the connection is created dynamically.

2.3.3. MOS Load Balancing

In MOS, load balancing is carried out by dynamic process migration [2]. As a result of the system architecture, a MOS process is not sensitive to its physical location: system calls which access resources that are not located on its current node are automatically forwarded by the linker to the remote node. The ease with which processes may migrate in the system allows dynamic load balancing to be implemented. In MOS, each node sends its own local load estimate to a set of randomly selected nodes every fixed amount of time (every second in the current implementation). Load estimates received from other processors are kept in a *load vector* and are "aged" to reflect their decreasing relevancy. Every processor may refuse to accept a migrating process if it so desires and each processor accepts at most one incoming process during one load balancing cycle. The load balancing algorithm may migrate a process which has used at least 1 second of CPU time on the local processor. Processes which have a history of "forking" new processes are given preference by the algorithm which chooses a process for migration. Also, processes with a history of I/O operations to some specific remote node are considered for migration to that node.

2.4. Related Systems

Distributed UNIX systems, in the broadest sense of the definition given at this chapter, are quite popular. In order to reduce the clarify the relative merits and weaknesses of MOS, the following gives a brief description of a handful of the more widely known systems which have some aspects in common with MOS.

The Masscomp RTU system provides users with a real time variation of UNIX [16]. It has a distributed file system which was built on a transaction based reliable data protocol. The Masscomp approach to the distributed file system is quite similar to the MOS approach: an *rinode* is used as a universal pointer to the inode of an allocated file. The RTU kernel, like the MOS kernel, routes standard UNIX system calls to a remote host when necessary. The system was not designed to allow process migration or load balancing.

The SUN distributed system, *SUNOS*, is built around a general purpose remote procedure call mechanism [17, 18]. Unlike MOS, where remote procedure calls are made only by the kernel, processes have access to the remote procedure mechanism. The SUN Network File System, or NFS, is implemented through use of kernel level RPC calls. NFS provides transparent access to remote file systems which may be "mounted" at arbitrary points in the file system tree. SUNOS incorporates heterogeneous machines into a distributed file system. As a result, it can not support process migration and load balancing.

In Sprite [11], as in MOS, processes can migrate dynamically. Each process has a unique home node which handles all location sensitive system calls, *e.g.* calls which return the current time. The home of a process is always the same as its parent process which means that each node must handle such system calls for its original processes and all of their descendents. As in the other systems mentioned, remote system calls are implemented through use of a remote procedure call mechanism. As in MOS, the RPC is a kernel to kernel mechanism. The RPC facility has been used to construct a transparent distributed file system. Universal pointers to open files are migrated with the process from one node to another. has a significant influence on the performance of the system.

Chapter 3
Performance of Remote System Calls

3.1. Introduction

System calls define the basic interface between the user and the operating system. The overhead associated with the remote execution of system calls has a significant influence on the performance of the system. In MOS, system calls can be divided into three groups: system calls with no remote implementation, system calls with *scall* versions which use the *funnel* mechanism and system calls with *scall* versions which do not use the *funnel* mechanism. After a brief description of the measurement techniques used, the chapter goes on to describe the performance of each of the types of system calls and the overall influence of remote system calls on a the performance of processes.

3.2. Measurement Technique

The remote system calls overhead was measured by running a set of benchmarking programs. Each benchmark measures the elapsed time required for local and remote execution of a system call that has a remote implementation. Each system call is executed 10,000 times on objects located on the local node and 10,000 times on remote objects. The system calls measured are all part of the standard UNIX interface. Some system calls were not measured, such as *mount* and *umount*, since these operations are relatively infrequent and the slowdown can be extrapolated from the results for similar system calls that were measured. The measurements were done when the remote and the local nodes were running in single user mode to reduce interference from uninvolved processes. As a side note, several system calls were carried out with up to 100,000 iterations. However, as reported in a related work by [9], the times do not change significantly as the number of iterations increase, so all measurements are based on the average time of 10,000 calls.

System Calls without a Scall Version

Several system calls in the MOS system have no need of a corresponding remote version since the call does not require access to objects at a location dependent site. For example, the standard "time" UNIX system call returns the current time. Since the time is synchronized between active MOS sites, this system call can always be handled by the kernel where the process is currently running. As a result, no performance penalty is paid for calling such a system call regardless of the location of the process.

System Calls with Funnels

Read and *write*, UNIX's input/output interface, are two of the most frequently used system calls. The *scall* versions use *funnels* to transfer the data to be read or written to and from a remote machine. In order to execute a single remote *read* system call, MOS does the following:

- The upper kernel routine, *read*, is called. It does elementary checking of the parameters, calls a kernel procedure which maps its file descriptor into a universal inode pointer, sets up its end of an input *funnel* and then passes the universal inode pointer and the other parameters to the local linker.
- The linker checks to see where the call needs to be executed. If it is a remote call, the linker encapsulates the parameters into a message and sends the message over the network to the target machine.
- On the remote machine, an ambassador process receives the message, sets up the remote end of the *funnel* and then executes the corresponding lower kernel procedure.
- A lower kernel procedure (*readl*) converts the universal inode pointer into a local inode and passes the inode to another lower kernel procedure, *readi*, which does the actual reading of the file.
- Until the requested amount has been read, *readi* reads one logical block at a time from the file and places the block in the *funnel* which causes the linker to encapsulate the block into a message and send the message over the network to the requesting node.
- The lower kernel finishes its work and passes control back to the ambassador. The ambassador encapsulates the return status of the call into a message and sends the message back to the requesting machine's linker.
- The linker and the upper kernel finish their part of the work on the requesting machine and pass the result back to the user.

In summary, the remote read causes the following messages to be sent over the network: one to initiate the call, one to return the exit status and one *funnel* message for each logical block read from the file. Due to the unreliable nature of the network, each of the messages sent also incurs the cost of an acknowledgement.

The results presented in Table 3.1 shows the cost of sequentially reading a regular file, using different buffer sizes. The second and third columns show the measured results of local and remote execution of the system call. The last column shows the *slowdown* factor which is obtained by dividing the remote execution time by the local execution time. This figure shows, as one would expect, that larger read requests suffer less when executed remotely.

Size Read	Elapsed Time (ms) for Local Read	Elapsed Time(ms) for Remote Read	Slowdown Ratio
2 bytes	2.20	10.00	4.55
8 bytes	2.33	10.40	4.46
32 bytes	2.40	10.60	4.41
128 bytes	3.20	11.80	3.69
512 bytes	7.20	15.80	2.19
1024 bytes	9.25	20.50	2.22
2048 bytes	14.23	37.48	2.63

Table 3.1 – MOS Read System Call

Note that the logical block size of the MOS file system is 1024 bytes. The above table shows that the elapsed time required to read one logical block from a remote file is approximately 11 milliseconds more than the time required to write one logical block to a local file, a slowdown of 2.22. Since the linker is forced to send at two packets for a 2048 byte buffer, the overhead increases to approximately 23 milliseconds and the slowdown factor increases to 2.63. The results for the write system call, not shown here, are similar with a slowdown ratio of 2.55 for a 1024 byte remote write.

In Table 3.2, a more detailed look at the read system call is presented. The measured execution times and the communication overhead incurred by the use of the *scall* mechanism for the *read* system call are shown. The second and fifth columns show the measured times for the local and remote execution of *read.*

Buffer Size	Elapsed Time (ms) for Local Read	Local RPC Overhead	Remote RPC Overhead	Elapsed Time (ms) for Remote Read
2 bytes	2.20	6.65	6.32	10.00
8 bytes	2.33	6.68	6.24	10.40
32 bytes	2.40	6.84	6.46	10.60
128 bytes	3.20	7.13	6.11	11.80
512 bytes	7.20	8.15	4.26	15.80
1024 bytes	9.25	9.48	4.83	20.50
1536 bytes	14.04	12.93	7.04	30.57
2048 bytes	14.23	15.23	9.54	37.48

Table 3.2 – Breakdown of Remote Read

The local and remote communication overhead times were measured by using two "load sink" processes. First, the amount of work performed by the load sink process was measured during a fixed time interval on each of the unloaded nodes. This was compared to the amount done while the benchmark was running. The cost incurred by each node was calculated to be the amount of milliseconds "stolen" from the load sink processes during the running of the benchmark. The remote RPC overhead is the amount incurred by the remote node minus the amount of time required to perform the actual system call. Therefore, in order to get an estimate of the remote communication overhead, the amount of time required to perform a local read system call was subtracted from the total measured amount. The result is slightly lower than the actual overhead since the processing of the upper kernel portion of the system call, done by the local node, is also subtracted from the remote cost.

The local node's overhead in a remote read involves executing the upper kernel portion of the read system call and the work done by its linker in initiating and managing the local end of the message passing. Similarly, the remote linker is responsible for most of the overhead paid by the remote node. The local overhead is quite similar to the remote overhead. The minimum elapsed time required for a remote *scall* is the sum of the *Maximum(local overhead, remote overhead)* and the *local execution time*. On the other hand, if the two processors execute their part of the overhead sequentially, the worst possible case, the remote execution elapsed time would be the sum of the three. In Table 3.3, the columns show the minimum expected remote execution time, the measured (elapsed) time required for execution of the *scall* and the worst expected execution time. The percentage of the parallelism achieved in handling the remote and local portions of the *scall* is computed from these.

Buffer Size	Minimum Predicted Time	Measured Remote Time	Maximum Predicted Time	Percent Parallelism Achieved
2 bytes	8.52	10.00	15.17	29.7
8 bytes	8.57	10.40	15.25	24.2
32 bytes	8.86	10.60	15.70	25.5
128 bytes	9.31	11.80	16.44	12.2
512 bytes	11.46	15.80	19.61	0
1024 bytes	14.08	20.50	23.56	0
1536 bytes	21.08	30.57	34.01	0
2048 bytes	23.77	37.48	39.00	0

Table 3.3 – Remote Read: Parallelism Achieved

There are several ways to reduce the overhead involved. First, the percentage of work done concurrently by the two machines involved should be increased as much as possible. This can be done by using the initial message packet sent over the network to "piggyback" some part of the data that needs to be sent. In this way, the remote linker can start transferring data immediately after receiving the request message. Another possibility is to lock the calling process in memory so that the networking hardware can transfer the data directly to the user memory space. Although the current MOS implementation does not incorporate these features since they require a greater consumption of main memory, the newest MOS system does incorporate similar overhead reducing measures. Additional reductions in overhead can be achieved by using hardware which provides a reliable delivery medium with checksums and automatic acknowledgements, thus relieving the processor of the burden.

An example of a system call that uses *funnels* is the *exec* call which uses *funnels* to transfer an executable file from a remote machine. In Table 3.4, the first job executed by the *exec* call has no data region and exits immediately. This demonstrates that the amount of time required to set up a new process is quite substantial. The second job has a data region the size of the standard UNIX text editor, *vi*, and also exits immediately. As the table shows, the overhead associated with the remote *exec* call is quite low.

Size of Process *exec*ed	Elapsed Time (ms) for Local Execution	Elapsed Time(ms) for Remote Execution	Slowdown Ratio
1,586 bytes	100.00	137.00	1.37
257,586 bytes	257.00	275.00	1.07

Table 3.4 – MOS Exec System Call

System Calls without Funnels

Most system calls involve only the transfer of the parameters and the return of the result, so that no use is made of the *funnel* mechanism. For example, the *namei* kernel routine performs one remote procedure call for each remote segment of a pathname. As a result, remote system calls like *access* and *stat* which spend a large portion of their execution time in *namei* pay heavily for remote execution, especially for files with long pathnames. In general, "short" system calls pay a heavier penalty in terms of percentages since the overhead is a substantial part of their run-time. Thus, short system calls like *lseek*, which simply changes the value of the offset pointer in a file descriptor structure, also pay a heavy penalty for remote execution. The "heavier" system calls such as *open/close* show the most favorable performance since the time required for local execution of the calls is already substantial. An extreme example of a heavy system call is *sync* which flushes the buffer cache associated with a certain device to disk.

Table 3.5 presents the measured results for commonly used system calls. Four system calls were measured in pairs: *open* and *close* were one pair and *link* and *unlink* were the second. In the first case, the two calls were paired due to the limitation on the number of open files allowed per process. In the second case, creating several thousand links, although possible, causes a severe degradation in performance as the number of links increase due to the linear search that must be performed on a directory. In both cases, the paired calls are often used together so the cumulative results are relevant for normal programming.

Name	Elapsed Time (ms) for Local Execution	Elapsed Time(ms) for Remote Execution	Slowdown Ratio
access	14.10	41.05	2.91
chmod	29.24	71.30	2.44
chown	26.26	71.26	2.71
dup	2.34	11.34	4.84
ioctl	1.66	9.62	5.80
link/unlink	66.70	133.95	2.01
lseek	1.15	5.72	4.97
open/close	37.00	83.00	2.24
stat	14.8	43.5	2.94
sync	12.35	16.35	1.32

Table 3.5 – MOS File Related System Calls

Table 3.6 presents the results for process related system calls. As noted for the file system related calls, the "lighter" system calls for processes are also heavily penalized in remote execution. Note that many process related system calls, including calls like *fork*, never involve remote resources and thus have no *scall* versions.

Name	Elapsed Time (ms) for Local Execution	Elapsed Time(ms) for Remote Execution	Slowdown Ratio
chdir	24.12	63.38	2.71
chroot	9.55	23.95	2.51
kill	4.10	7.70	1.88

Table 3.6 – MOS Process Related System Calls

3.3. Overall Performance of Remote System Calls

A reasonable estimate of the overall penalty that a process pays while executing remotely is a *weighted average*: the slowdown factor measured for each system call is multiplied by its relative use in a standard UNIX environment. The numbers used in Table 3.7 are based on statistics gathered in [11] for a standard 4.3bsd UNIX system for the most frequent system calls.

Name	Percentage of Calls	Has Scall Version	Effect (Percent * Slowdown)
read (1k bytes)	41.0	yes	91.0
close	17.5	yes	39.2
open	16.3	yes	36.5
write (1k bytes)	15.3	yes	39.0
fork	2.7	no	2.7
exit	2.7	no	2.7
exec	2.2	yes	3.0
create	1.2	yes	1.6
Total	98.9		240 percent

Table 3.7 – System Call Weighted Average

Since each process has a unique mix of system calls, the above result provides only a rough prediction of the system time slowdown. Many processes spend a major part of their execution time running in user mode which does not suffer a slowdown due to remote execution. Chapter 6 shows benchmark results from such applications that show an almost linear speedup.

Chapter 4
Interprocess Communication Mechanisms

4.1. Introduction

In order to utilize the processing power of a distributed system, a distributed application is broken into several subtasks which can run in parallel on different machines. On a UNIX system, such distributed applications have several drawbacks. The first drawback is the "heavy" nature of a UNIX process which typically takes several hundred milliseconds to create. One way to avoid this cost is to circumvent the UNIX process creation and handling mechanisms altogether by using *light weight processes* as is done in the MOS Distributed Lightweight Processes project [19]. Secondly, in addition to the time required for the computation of the task, each of a distributed application's subtasks must use some form of *interprocess communication* (*IPC*) mechanism to communicate with the other processes working on a given task. The overhead associated with IPC is a critical factor in determining the performance of a distributed application [14]. For example, if the amount of communication between processes is zero, as is the case with unrelated user processes on a general purpose system, distributed systems can gain impressive speedups. On the other hand, if a group of processes spend most of their time in communication and little time in computation, using a distributed approach can actually increase the overall time needed for computation.

In [20], Watson defines the following desired characteristics for a distributed system's IPC service:

- No apriori restrictions should be placed on which processes can communicate with which others. Processes should see the same IPC interface whether they are communicating with local or remote processes. Local host or network idiosyncrasies should be largely or completely hidden.
- Symmetric communication should be supported between equal and autonomous partners. Each partner must have full control over its interaction with the other, deciding when it is willing to communicate, how much of its resources it is willing to allocate to a given conversation and what events it is willing to be blocked on until they occur.
- Efficient transaction and stream oriented styles of communication should be supported.
- No restrictions should be placed on the lengths or contents of the basic meaningful mes-

sage.

- The basic IPC service should not limit or bias the direction of the higher level programming or application IPC may take.

In the MOS system, these subtasks are handled by separate processes which are spawned with the *fork* system call. The load balancing mechanism automatically distributes the processes between the available nodes which provides real parallel execution. This chapter analyzes the cost of two standard UNIX IPC mechanisms that the MOS system provides: pipes and AT&T's System V2 messages.

4.2. The UNIX Pipe Mechanism

In UNIX, a *pipe* is used to transfer data between related processes in a *first-in-first-out* manner (*FIFO*) [21]. Pipes are implemented as a pair of file descriptors which are returned by the *pipe* system call. All transfer of data to and from a pipe is done with the standard *read* and *write* system calls, regardless of the location of the communicating process. Writing to one of the file descriptors puts data into the pipe while reading from the other takes data out. Since UNIX mechanisms only allow sharing of open file descriptors between a process which opens the pipe and its children or between the children of such a process, the standard UNIX pipe mechanism does not provide a general purpose IPC mechanism. Pipes are finite: processes which try to write too much data (over 2048 bytes in MOS) into a pipe must block until the pipe has been emptied by a reader process taking out data from the other end. Since processes which try to read from an empty pipe are blocked until a writer process fills the pipes, the pipe mechanism can be used to synchronize processes. At the lowest level, pipes are built as a special type of file which means that it uses the standard lower kernel mechanisms for manipulating data buffers and inodes. In spite of their limitations, pipes are the only IPC mechanism that is available on every type of UNIX system. They provide several of Watson's [20] desired features in that the messages passed between processes are not restricted in size or content and the interface routines are not location sensitive. Pipes are quite useful in distributed applications since a parent process is typically used to spawn subtask processes, collect and then process the results of the subtasks' computations into a final answer.

4.3. The MOS Pipe Mechanism

The methods described in Chapter 3 to measure system calls were used to measure the performance of the pipe mechanism. The following table shows the elapsed time and the associated slowdown for sending messages of *size* bytes. An additional pipe was used to synchronize the reader and writer processes. The measured interval starts with the writer process sending the first message and ends when the receiving process reads the last message.

Size	Elapsed Time (ms) for Local Write	Elapsed Time(ms) for Remote Write	Slowdown Ratio
2 bytes	4.84	11.09	2.29
64 bytes	5.03	11.42	2.27
128 bytes	4.82	11.95	2.48
512 bytes	5.94	14.81	2.49
1024 bytes	6.55	18.47	2.82
2048 bytes	9.94	28.58	2.88

Table 4.1 – MOS Pipe Mechanism

The amount of time required to pass messages between two local processes, in the worst case, should be the sum of the time required to write the same size buffer to a regular file and the time required to read that size buffer from a file. For the smaller messages, the results are quite close to this worst case scenario since the write process writes uninterrupted until it fills the pipe. A context switch then occurs and then the reader process empties the pipe and the cycle starts again. Writing to a pipe proves itself to be more efficient than writing to a file as the size of the message increases, since the pipe mechanism does not write data to the disk or deal with indirect blocks which UNIX uses to implement large files. This result is shown in Table 4.2.

The remote test case involves a local writer process and a remote reader process. Both processes communicate through a pipe which is created on the writer process' machine. The worst performance would occur when both processes execute serially: the elapsed time would be equal to the sum of the required to perform a local write and a remote read (as shown in Chapter 3).

Size	Elapsed Time (ms) for Local Write	Elapsed Time(ms) for Local Read	Sum	Elapsed Time(ms) for Local Pipe
2 bytes	2.22	2.22	4.44	4.84
128 bytes	3.33	3.2	6.53	4.82
512 bytes	5.83	7.2	13.03	5.94
1024 bytes	8.92	9.25	18.17	6.55
2048 bytes	16.5	14.23	30.73	9.94

Table 4.2 – Comparison of Local Pipe and File I/O

As shown in Table 4.3, the remote pipe is considerably faster than this for all messages greater than two bytes in size. In fact, the performance of the pipe mechanism in the remote case is actually better than that of an equivalent read from a remote file in most of the cases.

Size	Elapsed Time(ms) Local Write	Elapsed Time(ms) Remote Read	Sum	Elapsed Time(ms) Remote Pipe
2 bytes	2.22	10.00	12.22	11.09
128 bytes	3.33	11.80	15.13	11.95
512 bytes	5.83	15.80	21.63	14.81
1024 bytes	8.92	20.50	29.42	18.47
2048 bytes	16.50	37.48	53.98	28.58

Table 4.3 – Comparison of Remote Pipe and File I/O

The MOS kernel treats *read* and *write* calls for pipes the same way it handles calls involving regular files. When the linker identifies a remote pipe, the data is encapsulated by the linker and sent through a funnel to the node which hosts the pipe. The gain in performance over I/O to a regular file is explained by the elimination of disk I/O: since the size of a pipe is limited, all of the data blocks are always in the buffer cache of the local node.

4.4 The Messages IPC Mechanism

A newer IPC mechanism, *messages*, is a standard feature of AT&T's UNIX System V.2 systems (see [21] for a detailed explanation). System V messages are a general IPC mechanism: any two processes can use them to communicate provided they know the *key* which uniquely identifies a queue of messages. The amount of data held in a certain message queue is limited by a system defined constant: sending a message to a full queue causes the writer process to receive an error code unless it specifically requests to sleep on the event of a reader process emptying the queue. In this way, the regular interface routines can remain location independent. Since a reader process has the option of waiting for an empty message queue to receive data, the message mechanism can also be used to synchronize processes. In order to allow *out of band* messages, messages are tagged with *types* and processes can send or read messages selectively from a given message queue according to type.

4.5. MOS Implementation of Messages

The MOS implementation of the messages mechanism is similar to the AT&T version. In MOS, each of the message interface routines maps into a single system call entry point. The parameters of the call are examined to determine which of the routines was called and which node is the home node of the message queue being used. The linker thens sets up a funnel for calls which have message data to transfer and calls the specific routine with its parameters on the host machine of the message queue. The specific routines executes and places any data into the funnel. Since the maximum size of the data is limited to a system defined maximum, the body of a message can always be passed through the funnel in one packet. A message queue can be migrated from the node of its creation to another machine by use of the *msgctl* with a MOS specific argument.

Size	Elapsed Time (ms) for Local Write	Elapsed Time(ms) for Remote Write	Slowdown Ratio
2 bytes	5.18	11.85	2.29
128 bytes	6.49	18.41	2.84
512 bytes	10.54	21.06	2.00
1024 bytes	15.97	27.43	1.72
2048 bytes	26.45	46.42	1.76

Table 4.4 – MOS Msg Mechanism

In MOS, as shown in Table 4.4, messages are a heavier mechanism than pipes. The period measured starts when the sending process starts sending and ends when the receiving process receives the last message. As with the "heavy" system calls measured in Chapter 3, the message mechanism's slowdown factor decreases as the size of the message increases. The slowness of the mechanism, almost twice as slow as the pipe mechanism, is offset by the increase in flexibility and sophistication that is provides. While the performance of the mechanism may be improved by eliminating memory copies from user space to kernel space, serious performance improvements in a general IPC mechanism like System V messages can only be achieved by the use of dedicated message sending hardware or by the use of a dedicated IPC coprocessor as was done in [22].

Chapter 5
Networking and Process Migration

5.1. Introduction

Dynamic process migration is one of the unique features of MOS. The MOS load balancing algorithms [2] use the dynamic migration mechanism to evenly redistribute processes from overloaded to less loaded machines. Distributed applications, like those described in greater detail in Chapter 6, simply create several subprocesses with the standard UNIX *fork* mechanism and the load balancing mechanism sees to the even distribution of the processes throughout the system. This chapter describes the performance of the funnel mechanism and then the speed of process migration.

5.2. MOS Networking Mechanisms

In any distributed system, performance is highly dependent on the speed of the physical network and the networking protocols implemented. In MOS, the user has no explicit access to the network. The linker uses the network to implement the following functions: process migration, funnels and remote kernel procedure calls. The performance of the remote kernel procedure calls was described in detail in Chapter 3.

Each of the mechanisms above has a separate implementation in the linker which handles network access. At the level of the linker, the data that needs to be transferred is broken into the appropriate sized blocks (the current implementation uses 1.5 Kbyte packets). The data is encapsulated into a message of the appropriate type, *e.g.* input/output funnel, various types of process data or *scall*. The separate implementations mean that there is a certain amount of duplication of code in the linker, but allows the handling of the networking to be tailored for the best performance of each type of data.

The lowest level of the data transfer is handled by a specific hardware driver, in our case the driver for the Pronet hardware. The Pronet driver is responsible for sending and receiving acknowledgements for the messages and guarantees reliable delivery of the messages. Again, the system is tailored to increase performance by using busy waits for small packet sizes and DMA for larger packets.

In general, all of the implementations perform the following four steps to transmit data on the network:

- A memory to memory copy is used to transfer user data from the user address space to the kernel space.
- The data and other message related data such as the message headers are copied into the Pronet hardware's physical memory buffer by a busy wait or a DMA transfer.
- A check sum is performed on the packet.
- The Pronet driver transmits the package over the network.

The reverse process is used by the receiving machine to copy the data from the remote machine to the local user data area. In sum, the optimal transfer rate can be roughly calculated as the sum of two memory to memory copies, two DMA copies, two check sums and the physical transmission time over the network. In addition, the amount of time in the upper kernel, context switching and the time required to prepare and send the packet that is required by the linker and the Pronet driver also have a considerable impact on the throughput. The speed of the physical network is determined by the hardware used: the Pronet token passing ring has a throughput of 10 Mbit/second (1,250 Kbytes/second). The following sections attempt to measure the effect that each of the other factors has on the network throughput.

5.3. The Funnel Mechanism

The funnel mechanism typifies the network related mechanisms. Therefore, analysis of the networking protocols centers on measurements of the data throughput of the funnel mechanism.

For the system call *read*, an input data funnel is created by the upper kernel portion of the read call. The linker may also create a new funnel before migrating a process, for example. After the funnel has been set up, the kernel uses a linker procedure call to send the data to a waiting light weight kernel process, an *ambassador*, which handles the remote end of the transaction.

The following table shows two of the desired measures: the time required for a local memory to memory copy and the throughput of an input funnel. Both cases show the measured throughput of the *read* system call while reading the MOS pseudo-device, **/dev/fun** which simply copies the requested number of bytes from an array of null characters into the funnel. In the local case, the linker implementation of the funnel simply performs a memory to memory copy of the data. In the remote case, the data is broken into packets of the size that can be sent over the network and then sent over the network. The amount of time spent in setting up the system call should be subtracted from the measured times, *e.g.* starting with the time that the user level call to *read* occurs up to the time that the linker starts sending data. Since it is not possible to measure this period exactly, the same measurements were taken with all of the calls to the linker "shorted" so that the upper level routines alone are called. Using this figure, the throughput at the level of the linker can be computed and is given in the fourth row of Table 5.1. A large 30,000 byte buffer size is read from the remote pseudo-file in order to amortize the time spent in the upper kernel between a large number of bytes.

Operation	Throughput (Kbytes/sec)
Local Memory to Memory Copy	1,163
Network Funnel Throughput	112
Process Migration Throughput	114.77
Network Funnel Throughput w/o Upper Kernel	115.9
Network Funnel Throughput w/o Checksum	164

Table 5.1 – Network Throughput

As the Table 5.1 also shows, local memory to memory copies are an order of magnitude faster than a network read. The throughput over the network was 112 Kbytes/second. The last row shows that compensating for the time spent in the upper kernel does not have a significant impact on the throughput, in fact it accounts for only 3 percent of the time spent in the transaction.

One attempt to improve the throughput involved removing the Pronet driver's checksum which is done on every message sent. Although the current network hardware requires that the error checking be done in software, future generations of hardware will be able to do this by themselves. An added advantage of not doing a checksum is a reduction in the number of message queues maintained by the Pronet driver (by one). Without the checksum, throughput was increased to 164 Kbytes/second, an increase of 46 percent. At this rate, a two machine transaction utilizes 13.12% of the Pronet's maximum bandwidth.

5.4. Process Migration

The process migration mechanism is built upon the remote kernel procedure call mechanism and the funnel mechanism. Remote procedure calls are used to request that a new procedure entry is set up at the remote site and to start execution of the migrated process on the remote machine after the migration completes. Funnels are used to transfer the procedure's text, data and stack regions. Not surprisingly, the throughput of process migration is closely tied to the performance of the funnel mechanism: the measured throughput is 114.77 Kbyte /second which is slightly better than that of the transfer rate of the funnel mechanism. This figure is an important parameter of the load balancing mechanism: the time it takes to migrate a process is taken into account when considering the possible benefits of migration.

Chapter 6
Distributed Applications

6.1. Introduction

The bottom line of performance for a distributed system is the performance of the distributed applications that run on the system. In the previous chapters, the slowdown associated with the various aspects of MOS were measured. In this chapter, the *speedup* of various applications is measured when they are run on the MOS system. The *speedup* of a distributed program is computed as the amount of time the application uses on a one processor system divided by the amount of time required on a multi-processor system. In the optimal case, a distributed application will have a linear speedup with an increasing number of processors, that is the application will run **k** times faster on a **k**-processor system than on a 1 processor system.

In MOS, applications benefit more or less from a multi-processor configuration depending on their nature. This chapter analyzes the performance of applications which are mixed I/O and CPU bound and almost pure CPU bound jobs.

6.2. Parallel Make

Make is a standard UNIX utility which is used to compile and maintain software systems. For large programs, make uses a *makefile* which lists the source files and the dependencies for each source file, how to compile the source file into an object file and how to link and load the object files. When make is run, it reads the makefile, generates the dependency tree and then forks a child process which executes the next unfulfilled goal, for example the compilation of the next source file of a software package.

The parallel version of *make* which runs on MOS, which allows a user to specify (in the makefile) which of a goal's dependencies may be prepared in parallel and the number of child processes which should be allowed to run concurrently at any one time. Each of the child processes runs either the C compiler, **cc**, the assembler, **as**, or the link-loader, **ld**. Most of the subgoals use the C compiler, so a short description of the compiler is justified.

The C compiler is a four phase compiler. Each instance of the program cc forks to create four child processes, each of which runs one of the four phases of the compilation. Each stage writes its results into a temporary file in the */tmp* directory. In new versions of the C compiler, *pipes* are used in place of the temporary files to transfer data between the different subprocesses that execute the various phases of compilation. Each of the phases of the compilation work in parallel to a limited extent.

The parallel make was used to compile the kernel of the MOS system which contains approximately 25,000 lines of code in 60 different source files. Parallel make represents a mixed IO/CPU bound job since each of the four phases of the compilation reads one source file, computes its stage of the compilation and writes its result into a new file or the final object file. The parallel version has two serial phases: the generation of the dependencies and the linking and loading of the compiled object files.

Single Machine Compilation

The first run of the parallel *make* was run on a single MOS node which did not allow processes to migrate. In Table 6.1 we give results that show as the number of processes increases, the overall time for the compilation of the kernel decreases since a better utilization of the CPU is achieved as one compilation uses the CPU while another sleeps on the completion of an IO request. If the percentage of time that each child process sleeps is great enough, there is little contention for the CPU. However, since each of the compilations requires a relatively heavy amount of processing the optimal number of processes running in parallel on one machine is low.

Processes	User Time	System Time	Elapsed Time	CPU Utilization
1	622.95	288.10	1161.00	78.00
2	628.60	292.55	1042.00	88.00
3	635.05	296.75	987.00	94.00
4	636.05	298.50	995.50	93.00
5	636.10	302.10	1006.00	93.00
6	645.95	317.30	1031.00	93.00
7	646.80	313.35	1014.50	94.00
8	649.80	323.00	1040.50	93.00
16	658.13	327.28	1056.25	93.00

Table 6.1 – Parallel Make with 1 Processor

The table shows that the parallel make gets a slight speedup as the number of processes increases to 3 and a slowdown with more than 3 processes. In fact, with around 17 processes the resources of the single machine, *e.g.* the number of slots in the system's process table, are exhausted and the parallel make fails. As would be expected, increasing the number of processes causes increased context switching between the processes that run concurrently. This in turn increases the amount the parallel make spends in system mode. A steady increase in the time spent in user mode was also observed. This is largely a result of the way in which UNIX times a process: one *clock tick* of time is added to its user mode time if it is running in user mode when the clock interrupt occurs. This does not mean that the process was running during the entire time slice attributed to it: if a system interrupt occurs during the previous time slice, the scheduler is called and may choose to let a new process run. That has two effects: charging the time spent by the preempted process and the interrupt handling routine against the user time of the process that is running at the end of the time slice and letting the pre-empted process get a bit of "free" processing time. On the whole, the amount of time unjustly charged against processes evens out, but the interrupt handler's time ends up being charged, at least in part, as the user time of some process.

Multiple Machine Compilation

The next benchmark measures the speedup gained by letting the processes freely migrate between two MOS nodes. Table 6.2 shows the results:

Processes	User Time	System Time	Elapsed Time	CPU Utilization
1	625.20	287.35	1146.00	79.00
2	631.00	343.90	969.50	100.00
3	634.00	439.90	851.50	125.50
4	632.40	463.85	813.00	134.50
5	636.35	484.60	766.50	146.00
6	633.75	491.60	741.50	151.50
7	634.70	516.20	739.00	155.50
8	636.80	504.25	739.50	153.50
16	636.90	512.30	758.00	151.00

Table 6.2 – Parallel Make with 2 Machines

With two processes running in parallel, the best speedup relative to the one machine case happened with 9 processes running in parallel, a speedup of 1.57. Another interesting measure of the parallelism achieved is the CPU utilization which shows the utilization relative to a one processor. In the two machine case, the utilization reaches around 156% of a single processor.

The measured speedup with three and four machines is only slightly better than the two machine case, due to the high rate of IO that is done with the disk of the host machine of the parallel make which must handle the processing of the remote IO requests from all of the other machines. Table 6.3 shows the results for the 4 machine case.

Processes	User Time	System Time	Elapsed Time	CPU Utilization
1	624.10	275.60	1213.00	73.50
2	631.05	339.50	940.50	103.00
3	633.40	474.75	828.50	133.00
4	637.95	503.20	758.00	150.00
5	642.00	537.65	732.50	161.00
6	638.05	541.05	697.50	168.50
7	638.45	543.90	686.00	172.00
8	641.35	549.55	667.00	178.00
16	645.55	556.40	698.00	172.00

Table 6.3 – Parallel Make with 4 Machines

Here the best case is with 8 processes running in parallel, only two compilations per node. The speedup achieved over the 1 machine run was 1.48 while the speedup relative to the best 2 machine run is only 1.09. There are several reasons for the poor scalability of the parallel make. Firstly, the amount of IO that is processed by the host machine is an intrinsic bottleneck for a parallel computation. The second reason for the poor increase in the 4 machine speedup is the two serial phases of the parallel make itself: the startup phase which generates the dependency tree and the linking and loading phase. Table 6.4 shows the measured times with these two phases.

Stage	User Time	System Time	Elapsed Time
Startup	15.80	10.13	27.00
Linking	45.43	27.33	97.60
Total	61.23	37.46	124.60

Table 6.4 – Timing of the Serial Phases of Make

If the serial times are subtracted from the best time for the 1 machine case, the elapsed time for the parallel phase is 862.4 seconds. Similarly, the best time with the two and four machine runs are 603.9 and 542.4 seconds respectively, which is a 1.59 speedup. When measured this way, the 4 machine speedup is around 40% of the optimal speedup of 4.0.

In order to improve the speedup of the four machine case even further, the C compiler was changed to randomly choose one of the four machines as the site of the temporary files that it creates. Table 6.5 shows the results of this run.

Processes	User Time	System Time	Elapsed Time	CPU Utilization
2	634.25	425.60	928.00	114.00
3	639.05	480.95	753.50	148.50
4	634.25	504.75	643.00	176.50
5	645.15	545.15	628.50	189.00
6	642.30	533.20	569.50	205.50
7	645.60	541.85	544.50	217.50
8	646.60	540.90	512.50	231.00
9	646.55	551.75	508.00	235.50
10	640.43	530.38	486.25	240.25
11	653.55	547.30	489.50	245.00
12	648.72	533.30	462.00	255.40
16	654.90	573.50	495.50	248.00
32	663.28	583.38	489.25	254.50
64	663.20	585.30	488.75	255.00
72	663.70	581.20	493.00	252.00

Table 6.5 – Make with Four Machines and Random Temporary Files

As the table shows, the best case measured took 462 seconds (without the serial phases, 337.4 seconds). This is a speedup of the parallel phase relative to a 1 machine case of 2.56 which is 64% of the optimal and a speedup relative to the best case from the unmodified make program of 1.61. A very slight improvement is achieved by distributing the source files as well across the machines, but due to the artificial nature of the method, the results are not included in this report.

The results for a run with a random distribution of files across 5 MOS nodes is given in Table 6.6. This produced a minimum run time of 415.75 seconds (291.15 seconds of parallel computation) with the speedup of the parallel phase being 2.96, 59% of the optimal. This is slightly less than with the four machine percentage due to the problematics of scaling up an IO bound application.

Processes	User Time	System Time	Elapsed Time	CPU Utilization
1	628.38	443.52	1355.00	78.75
2	643.50	501.58	977.25	117.00
4	644.53	524.28	647.00	180.00
8	651.40	555.88	495.25	243.25
16	660.30	574.28	432.25	285.25
32	664.88	585.05	420.25	297.25
40	666.33	593.95	418.75	300.50
50	660.00	575.42	415.75	296.75
64	665.23	579.37	421.00	295.00

Table 6.6 – Make with Five Machines and Random Files

6.3. A Parallel Version of the Traveling Salesman Problem

The traveling salesman problem requires an extensive tree search of its problem space. The implementation of the traveling salesman problem [12] that is measured in this section is basically a CPU bound job with some IO operations. One master process forks the requested number of children who then loop endlessly waiting for a message from the master indicating what part of the problem space they should search. At the end of each stage, each of the children pass back the two best paths that they find. The returned paths are sorted by the master process who keeps a constant *number of results* best paths (r, determined by a parameter), and uses them as the base paths for the next stage. The number of stages is equal to the number of cities (n) in the graph. Since each of the children does a tree search of a fixed depth of four at each stage of the computation, the complexity of the problem is O(r n sup 5).

Table 6.7 shows an instance of the problem run with 12 cities and 100 results on a single machine. Unlike the parallel make measured in the previous section, no speedup is achieved by using a number of processes on one machine since the job is purely CPU bound and the processes do not often give up the CPU willingly.

Processes	User Time	System Time	Elapsed Time	CPU Utilization
1	60.9	3.7	65	99
2	63.8	5.0	70	98
3	67.6	4.4	74	97
4	73.0	8.0	91	89
5	73.3	8.0	109	74

Table 6.7 – One Node Traveling Salesman with 12 Cities and 100 Results

Table 6.8 shows the results when the same problem is run on an increasing number of machines. Since the entire computation finishes in around one minute, the MOS load balancing mechanism is slow to move processes away from the original machine. However, the mechanism does migrate processes which chronically *fork* new processes, a heuristic which has the side benefit of spreading the load quickly between available nodes when a larger number of processes is used. All in all, this size of problem is not heavy enough to truly benefit from the distributed nature of MOS and the best speedup obtained is 2.91 on all seven machines, 41% of the optimal speedup.

Number of Machines	Processes	Elapsed Time	CPU Utilization	Speedup
1	1	65	99	1.0
2	2	44	168	1.48
4	6	42	219	1.55
7	7	35	288	1.86

Table 6.8 – Traveling Salesman with 12 Cities and 100 Results

Table 6.9, shows the effect of increasing the number of cities to 14 and leaving the number of results passed between stages at 100. The predicted increase in complexity is 2.15 times. The effect on the one process case shows the elapsed time was 2.28 times the elapsed time for the previous case, a close fit to the complexity predicted above (95%). In this case, the subtasks have more time to run on the available nodes after migration and the corresponding speedup obtained is 3.08, 44% of the optimum.

Number of Machines	Processes	Elapsed Time	CPU Utilization	Speedup
1	1	148	99	1.0
2	2	89	186	1.66
4	4	58	300	2.55
7	7	48	393	3.08

Table 6.9 – Traveling Salesman with 14 Cities and 100 Results

Table 6.10, shows a run with 24 cities and 200 results, a 64 fold increase in expected run time. With this example, each subtask runs for a long enough time to dwarf the startup time required and the time required to pass messages. The speedup measured with all seven machines is 6.19, 88% of the optimum speedup possible.

Number of Machines	Processes	Elapsed Time	CPU Utilization	Speedup
1	1	5462	100	1.0
2	6	2863	197	1.91
4	4	1447	389	3.77
7	7	882	647	6.19

Table 6.10 – Traveling Salesman with 24 Cities and 200 Results

The utilization of the processors involved also reached 92 percent of the available power. The eight percent of the combined CPU power that was not utilized can be reduced by changing the implementation in a way that reduces the time each subtask sleeps while awaiting a job assignment from the master process, for example by sending each task all of its jobs in a single message at the beginning of a computational round. Another way to reduce the unused CPU time is to manually place the subtasks at a certain node instead of letting the MOS mechanism distribute them. With foreknowledge of the length of the computation required for each of the subtasks, the initial quantum of processing required of a process before it becomes a candidate for migration can be side-stepped allowing each subtask to immediately start processing on a different node.

6.4. Conclusions

While the parallel make does not scale up as well as other types of problems due to its IO bound nature, the speedups gained with 4 and 5 machine configurations of the MOS system is quite impressive. A more innovative approach to file IO, such as is suggested in [23], could have a considerable impact on the parallelization of this type of problem. As a side note, the average length of time required for processing of a subgoal was 20.34 seconds while 3 of the subgoals required more than 65 seconds. Without dynamic process migration, the large deviation from the average could result in a much poorer speedup.

The traveling salesman problem is more suited to distributed computing than the parallel make, but still has some wasted CPU cycles during its message passing phase. While the algorithm can be improved in a few ways, the speedups obtained are consistently good with an increasing number of nodes, provided the problem is of sufficient complexity. The percentage of the optimal speedup decrease slightly as the number of machines increased from 2 to 7, from 96 percent down to 88 percent respectively. This could be the sign of a problem with scaling up some distributed mechanism of the system, but without a much larger number of processors on which to run tests it is not possible to determine.

Chapter 7
Conclusions

The purpose of this work was to measure the performance of the MOS multicomputer system and several distributed applications. One of the most significant outcomes of this work is pointing out weak areas of the system and providing suggestions for improvement. The slowdown factor associated with the remote kernel procedure call is low when compared with the the overhead of the Sprite system's remote calls, but there is room for improvement of the implementation. Firstly, remote calls which use the funnel mechanism should make use of "piggyback" techniques to transmit the initial funnel data in the initial request packet (or the acknowledgement packet returned by the remote node in the case of the remote read). This can reduce the overhead of calls which involve small amounts of data considerably. Secondly, a further reduction can be attained by copying the user data directly from user space to the network hardware buffers and visa versa. In this case, the slowdown factor would be reduced for all of the calls which involve data transfer.

There is room for improvement in the implementation of AT&T's System V.2 messages. As seen in the measurements of the parallel make, distributing the IO randomly between machines brought a significant improvement in the speedup. Accordingly, distributing this general message passing system should improve the performance of jobs that do heavy message passing. There are two problems with a "distributed" V.2 message mechanism: first of all, the mechanism was designed for a single machine and its namespace is flat, that is each of the message queues is described by a single integer. This would have to be changed to allow for specification of the host machine when opening an existing message queue or creating a new one. The second problem is one of garbage collection: since the mechanism is a general purpose one, all of the message queues are left in memory after the demise of the creating process unless specifically removed. Even for a single machine, this approach can rapidly consume the queues if unruly processes do not take care to remove queues when finished. Since the kernel has no way of knowing if a queue is going to be used in the future by a yet uncreated process, it can not perform garbage collection on the message queues. Distributing this mechanism only multiplies the problem. One way of side-stepping this problem is to distribute only the "private" queues, which are removed after the death of the creating process. This type of mechanism is not as general since it allows communication only between parent process and its descendents, but it would be appropriate for many types of distributed applications.

The implementation of the MOS Pronet driver, the lowest level of the networking, could be improved significantly by removing the checksum done on packets if more sophisticated hardware could be used. Also, the size of the packet passed on the network could be increased which in order to allow for the "piggybacking" mentioned above. The process migration mechanism is tightly tied to the lower level mechanisms and would benefit from improvements in their implementations.

Another outcome of this work is the analysis of the suitability of various distributed applications to the MOS environment. Not surprisingly, significant speedups are obtained only by processes which do little interprocess communication and are at least moderately long lived. While this describes a large class of difficult problems, such as the Traveling Salesman, some problems, like sorting algorithms, involve too much IO to obtain really impressive speedups on a multi-machine configuration. While the current system did not allow testing of algorithms on more than seven nodes, the speedup associated with a the Traveling Salesman increased roughly linearly with the number of processors.

In conclusion, MOS was found to provide a reliable and relatively good performance at the level of the kernel mechanisms and, in the best case, a linear speedup with an increasing number of processors.

References

[1] A. Barak, A. Litman, MOS: A Multicomputer Distributed Operating System, *Software - Practice and Experience* 15(8), Aug. 1985, 725-737.

[2] A. Shiloh, Load Sharing in a Distributed Operating System, Dept. of Computer Science, The Hebrew University of Jerusalem, M.S. Thesis, July 1983.

[3] D. Bar, File Replication in the MOS Multiple Computer System, M.S. Thesis, Dept. of Computer Science, The Hebrew University of Jerusalem, Sept. 1986.

[4] A. Barak, O. G. Paradise, MOS - Scaling Up UNIX, *Proc. Usenix Technical Conference*, Atlanta, GA, June 1986, 414-418.

[5] R. Hofner, The MOS Communication System, M.S. Thesis, Dept. of Computer Science, The Hebrew University of Jerusalem, Feb. 1984.

[6] A. Barak, D. Malki, R. Wheeler, AFS, BFS, CFS, ... or Distributed File Systems for UNIX, *Proc. EUUG Workshop on Dist. Systems*, Manchester, England, Sept. 1986.

[7] S. Leffler, M. J. Karels, M. K. McKusick, Performance Improvements and Functional Enhancements in 4.2BSD, *Proc. Usenix Technical Conference*, Portland, OR, June 1984, 237-252.

[8] M. J. Karels, M. K. McKusick, Performance Improvements and Functional Enhancements in 4.3BSD, *Proc. Usenix Technical Conference*, Portland, OR, June 1985, 519-531.

[9] L. F. Cabrera, M. J. Karels, D. Mosher, The Impact of Buffer Management on Networking Performance in Berkeley UNIX 4.2BSD: A Case Study, *Proc. Usenix Technical Conference*, Portland, OR, June 1985, 507-517.

[10] L. F. Cabrera, E. Hunter, M. J. Karels, D. A. Mosher, User-Process Communication Performance in Networks of Computers, *IEEE Transactions on Software Engineering*, Jan. 1988.

[11] F. Douglis, J. Ousterhout, Process Migration in the Sprite Operating System, *Proceedings of the Seventh International Conferenence On Distributed Computing Systems*, Berlin, Sept. 1987, 18-25.

[12] S. Lin and B. W. Kernighan, An Effective Heuristic Algorithm for the Traveling-Salesman Problem, Operations Research, Vol 21, 1973, 498-516.

[13] J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Theil, Locus – A Network Transparent, High Reliability Distributed System, *Proc of the 8th SOPS 15(5)*, Pacific Grove, CA., Dec. 1981, 169-177.

[14] M. L. Scott, A. L. Cox, An Empirical Study of Message-Passing, *Proceedings of the Seventh International Conferenence On Distributed Computing Systems*, Berlin, Sept. 1987, 536-543.

[15] S. Zhou, D. Ferrari, A Measurement Study of Load Balancing Performance, *Proceedings of the Seventh International Conferenence On Distributed Computing Systems*, Berlin, Sept. 1987, 490-497.

[16] C. T. Cole, P. B. Flinn, A. B. Atlas, An Implementation of an Extended File System for UNIX, *Proc. Usenix Technical Conference*, Portland, OR, June 1985, 131-149.

[17] B. Lyon, Sun Remote Procedure Call Specification, *Sun Microsystems Technical Report*, 1984.

[18] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, B. Lyon, Design and Implementation of the Sun Network File System, *Proc. Usenix Technical Conference*, Portland, OR, June 1985, 119-130.

[19] D. Malki, Distributed Lightweight Processes in MOS, M.S. Thesis, Dept. of Computer Science, The Hebrew University of Jerusalem, June, 1988.

[20] R. W. Watson, IPC Interface and End-to-End (Transport) Protocol Design Issues, *Distributed Systems Architecture and Implementation*, B.W. Lampson (ed.), Springer-Verlag, Berlin, 1981, 140-174.

[21] M. J. Bach, The Design of the UNIX Operating System, Prentice-Hall, 1986.

[22] J. W. Wendorf, H. Tokuda, An Interprocess Communication Processor: Exploiting OS/Application Concurrency, Technical Report, Carnegie Mellon University, March 1987.

[23] A. Barak, B. Galler, Y. Farber, A Holographic File System for a Multicomputer with Many Disk Nodes, *Technical Report 88-6*, Dept. of Computer Science, The Hebrew University of Jerusalem, May 1988.